TinkerType: A Language for Playing with Formal Systems

Michael Y. Levin
Department of CIS
University of Pennsylvania
milevin@cis.upenn.edu

Benjamin C. Pierce
Department of CIS
University of Pennsylvania
bcpierce@cis.upenn.edu

November 22, 2005

Abstract

TinkerType is a framework for compact and modular description of formal systems (type systems, operational semantics, logics, etc.). A family of related systems is broken down into a set of *clauses*—individual inference rules—and a set of *features* controlling the inclusion of clauses in particular systems. Dependency relations on both clauses and features and a simple form of "judgement signatures" are used to check the consistency of the generated systems.

As an application, we develop a substantial repository of typed lambda-calculi, including systems with subtyping, polymorphism, type operators and kinding, computational effects, and dependent and recursive types. The repository describes both declarative and algorithmic aspects of the systems, and can be used with our tool, the TinkerType Assembler, to generate calculi either in the form of typeset collections of inference rules or as executable ML typecheckers.

1 Introduction

The quest for modular presentations of families of programming language features has a long history in the programming languages community. At the semantic level, language designers since Landin [Lan65, Lan66] have understood how to view a multitude of high-level constructs through the unifying lens of the lambda-calculus. Further work has led to more structured approaches such as categorical semantics (e.g. [Gun92, Mit96, Jac99]), action semantics [Mos92], and monadic frameworks [Mog89]. Using these tools, it is now possible to synthesize many different interpreters [Ste94, LHJ95, Esp95, etc.] and compilers [LH96, HK98, etc.] from common blueprints or interchangeable building blocks.

On the syntactic level (the formal systems used to define typechecking, operational semantics, and program logics), progress on unifying formalisms has been slower. There have been some significant achievements in restricted domains, including Barendregt's Pure Type Systems [Bar92] and Sulzmann, Odersky, and Wehr's generic treatment of type inference for systems of constrained types [SOW97]. (A related result outside the domain of programming languages is Basin, Matthews, and Viganò's modular presentation of modal logics in Isabelle [BMV95].) In of these proposals, the idea is to define a single "parameterized" system from which many particular systems can be obtained by instantiation. This method supports once-and-for-all proofs of properties like subject reduction and decidability that apply automatically to all instances. However, to give a single, parametric description of a collection of formal systems, we must first understand *all* the possible interactions among their features. If some combinations of features are not well understood, a less structured (and hence more flexible) approach is required.

Our goal is to develop a framework that facilitates compact and modular description of very diverse collections of formal systems (e.g., all known typed lambda-calculi). We adopt a *feature-based* approach, breaking down a family of formal systems into a set of *clauses*—individual inference rules—plus a set of *features* equipped with a *dependency* relation. A clause may have multiple variants, each annotated with a

set of *relevant features* that control its inclusion in particular systems. A complete system is specified by a set of features.

Several things can go wrong in the process of maintaining the repository of features and clauses and extracting systems from it. A change in a clause may introduce inconsistencies with other variants of the same clause; a set of features identifying a system may be nonsensical; the clauses of a system may turn out to be incompatible with each other. In our open-ended setting, ensuring the "reasonableness" of generated systems is a difficult problem. (In particular, our approach is less useful for generic reasoning about families of systems than the parametric approach. This is the price we pay for breadth of coverage.) We have, however, identified several common sources of error in practice and introduced static consistency checks to prevent them.

The contributions of this work are twofold. First, we formalize the TinkerType language and describe its implementation. Second, we use this framework to construct a taxonomy of a number of familiar typed lambda-calculi, including systems with subtyping, polymorphism, type operators and kinding, computational effects, and dependent and recursive types. We specify both declarative and algorithmic aspects of the systems and present extracted systems either in the form of typeset collections of inference rules or as executable ML typecheckers. This repository of type systems is useful in itself (it forms the skeleton of a forthcoming book [Pie]) and gave us substantial experience with using the TinkerType framework in practice.

The remainder of the paper proceeds as follows. In Sections 2 and 3, we give precise definitions of clauses, features, and the process of composing systems and checking their consistency; Section 4 describes our implementation. Section 5 presents our collection of typed lambda-calculi. Sections 6 and 7 discuss related and future work.

2 Assembling Systems from Features and Clauses

A formal system can be described as a set of *judgements*, each consisting of a set of *clauses*. The simply typed lambda-calculus (STLC), for example, is a formal system with two judgements: typing and evaluation. The typing judgement contains clauses like

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_2{\to}\mathtt{T}_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_1}$$

while the evaluation relation has clauses like the beta-reduction rule.[1]

This is obviously a rather syntactic view of formal systems. More abstractly, we might say that the simply typed lambda-calculus is a pair of sets of derivation trees: one set of trees with conclusions like $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and one with conclusions like $\mathtt{t} \to \mathtt{t}'$. More abstractly yet, we might view the STLC as a pair of *relations* obtained from these sets of trees. Alternatively, the STLC can be represented by a pair of functions in, say, ML. Since we are interested in all of these views, we avoid committing to a particular one by taking clauses as primary and dealing with them in our formalism.

Clauses may appear in many different systems. For example, both the pure STLC and the STLC with booleans contain the application rule shown above. On the other hand, in other systems, clauses may take different forms. In (the algorithmic presentation of) of the STLC with subtyping, the application rule refines the rule above by adding an extra subtyping premise.

features control both which clauses and which version of each clause

dependency

a repository comprises a set of features, a dependency relation on features, and a set of clauses, each annotated with a set of relevant features.

To put a little more flesh on these bones, let us examine a small repository involving type systems with features for booleans, arrow types and subtyping. For the sake of brevity, the clauses we present will only implement the algorithmic typing and subtyping judgements.

[1] Strictly speaking, there are also "judgements" defining the syntax of types, terms, and contexts. For example, the term syntax judgement contains clauses like "if $\mathtt{T}_1$ and $\mathtt{T}_2$ are types, then so is $\mathtt{T}_1{\to}\mathtt{T}_2$." Our discussion elides these syntactic judgements, for brevity.

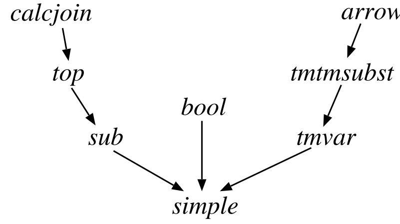


Figure 1: Features for arrows, booleans, and subtyping

Figure 1 shows the features needed to build systems with arbitrary combination of the above options and the dependencies between them. The features *bool*, *sub*, and *arrow* represent the choices of booleans, subtyping and arrow types mentioned above. The feature *top* introduces the top type and the subtyping rule for it. The feature *calcjoin* represents the functionality of calculating joins and meets of types. The presence of this feature signals that the subtype relation is closed under meets and joins, i.e., that we can calculate the least upper bound of any pair of types. The technical features *tmtmsubst* and *tmvar* stand for representation of variables and operations on them. Finally, the root feature *simple* is a common ancestor to all the features related to the typing judgement.

Figure 2 defines the set of clauses. To complete the definition of the repository, we define a refinement relation in which the contents of the T-App $[\mathit{arrow}, \mathit{sub}]$ and T-If $[\mathit{bool}, \mathit{calcjoin}]$ clauses refine the contents of the T-App $[\mathit{arrow}, \mathit{simple}]$ and T-If $[\mathit{bool}, \mathit{simple}]$ clauses respectively.

We can build the following systems from this repository:

- $[\mathit{bool}, \mathit{simple}]$: first-order boolean expressions
- $[\mathit{arrow}, \mathit{simple}]$: simply typed lambda-calculus
- $[\mathit{arrow}, \mathit{bool}, \mathit{simple}]$: simply typed lambda-calculus with boolean expressions
- $[\mathit{arrow}, \mathit{sub}]$: simply typed lambda-calculus with subtyping
- $[\mathit{arrow}, \mathit{bool}, \mathit{calcjoin}]$: simply typed lambda-calculus with subtyping and booleans

The last system in the list is the most extensive system one can build from the given repository. It contains every clause except T-App $[\mathit{arrow}, \mathit{simple}]$ and T-If $[\mathit{bool}, \mathit{simple}]$, which are superceded by T-App $[\mathit{arrow}, \mathit{sub}]$ and T-If $[\mathit{bool}, \mathit{calcjoin}]$.

We formalize these intuitions by specifying a *repository* to be a tuple $\langle FTS,\ D,\ C,\ CLS\rangle$, where:

- FTS set of features,
- $D \subseteq \mathcal{P}(FTS) \times \mathcal{P}(FTS)$ is a mapping between sets of features that represents dependencies between features,
- C is a set of clause contents,
- CLS is a set of clauses.

We define the function *implied* that takes a set of features and enriches it with all the features implied by the features in the given set as follows:

$$b \in \mathit{implied}(F) \text{ if } b \in F \text{ or } b \in D(F') \text{ for some } F' \subseteq F.$$

Now we introduce a function *closure* as the least fixed point of the function implied. Finally, we say that a set of features F_1 *dominates* another set F_2 if $\mathit{closure}(F_2) \subseteq \mathit{closure}(F_1)$.

$$\frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \quad \text{T-Var } [\mathit{tmvar}, \mathit{simple}]$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_2 \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \vdash \lambda\mathtt{x}{:}\mathtt{T}_2.\mathtt{t}_1 : \mathtt{T}_2 \rightarrow \mathtt{T}_1} \quad \text{T-Lam } [\mathit{arrow}, \mathit{simple}]$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_2 \rightarrow \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_1} \quad \text{T-App } [\mathit{arrow}, \mathit{simple}]$$

$$\Gamma \vdash \mathtt{false} : \mathtt{Bool} \quad \text{T-False } [\mathit{bool}, \mathit{simple}]$$

$$\Gamma \vdash \mathtt{true} : \mathtt{Bool} \quad \text{T-True } [\mathit{bool}, \mathit{simple}]$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Bool} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T} \qquad \Gamma \vdash \mathtt{t}_3 : \mathtt{T}}{\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}} \quad \text{T-If } [\mathit{bool}, \mathit{simple}]$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_2 \rightarrow \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{U} \qquad \mathtt{U} <: \mathtt{T}_2}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_1} \quad \text{T-App } [\mathit{arrow}, \mathit{sub}]$$

$$\frac{\mathtt{T}_1 <: \mathtt{S}_1 \qquad \mathtt{S}_2 <: \mathtt{T}_2}{\mathtt{S}_1 \rightarrow \mathtt{S}_2 <: \mathtt{T}_1 \rightarrow \mathtt{T}_2} \quad \text{S-Arrow } [\mathit{arrow}, \mathit{sub}]$$

$$\mathtt{S} <: \mathtt{Top} \quad \text{S-Top } [\mathit{top}]$$

$$\mathtt{Bool} <: \mathtt{Bool} \quad \text{S-Bool } [\mathit{bool}, \mathit{sub}]$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \qquad \mathtt{T}_1 <: \mathtt{Bool} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2 \qquad \Gamma \vdash \mathtt{t}_3 : \mathtt{T}_3 \qquad \mathtt{T} = \mathtt{T}_2 \vee \mathtt{T}_3}{\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}} \quad \text{T-If } [\mathit{bool}, \mathit{calcjoin}]$$

Figure 2: Clauses for arrows, booleans, and subtyping

A *clause* cl is a triple $\langle n,\ F\ c\rangle$, where n is a label identifying the clause, F is a set of features that governs inclusion of the clause in particular systems, and c is the actual content of the clause. We say that cl is *relevant* to the set of features F.

Now we have the tools to specify how a system is assembled given a set of features F. First, extract from the repository all the clauses whose sets of features are dominated by F. Then, partition the obtained clauses into sets of clauses with identical labels. Verify that each partition has a single maximal clause according to the dominance relation or, if there are multiple maximal clauses in a partition, that all of them have equivalent contents. Select a maximal clause from each partition. The contents of these clauses form the system.

3 Consistency Checking

[recall the two T-App clauses. Obviously, one is a *refinement* of the other. If we edit the first one, we want to check...]

In this section, we describe how to augment the formalism presented in the previous section with a simple type system that allows us to check consistency of generated systems. We already described one

consistency check between the refinement and dominance relations on clauses. This ensures that the user's expectations about the hierarchy of clauses and the way clause contents override each other reflect the actual dependencies between clauses generated from their feature annotations and the dependencies between features. This section introduces some additional consistency checks whose purpose is to ensure that no "stray" clauses are included in systems by accident, and that and that the clauses that are included have "compatible signatures." [,,, This is only partially implemented at the moment. See the next section.]

First we add the following elements to the repository of the previous section: $\langle \ldots R, FC, S, J, JC, CS \rangle$ where

- R is a binary refinement relation on C, and
- FC is a set of *feature constraint formulas*,
- S is a set of labels representing *syntactic categories*,
- J is a set of labels representing *judgement forms*,
- $JC \in J \rightarrow \mathcal{P}(FTS)$ is a *judgement enabling* function, and
- CS is a *clause signature* relation (its type is given below).

Clause contents form a set with an equivalence and partial order refinement relations over it. The intuition behind the latter is, roughly speaking, that one clause content refines another if it carries more information than the content of the other clause. We say that one clause refines another if its content refines that of the other clause. (For now, we assume that the refinement relation is given to us externally as part of the repository; the TinkerType tool provides a concrete way of calculating the refinement relation.)

Consider the following sample clause.

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Bool} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T} \qquad \Gamma \vdash \mathtt{t}_3 : \mathtt{T}}{\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}} \qquad \text{T-IF } [\textit{bool}, \textit{simple}]$$

This is a typing rule labeled T-IF. The clause is relevant to the combination of features *bool* and *simple*, and that implies that whenever we build a system of simple types with booleans, the above rule (or some refinement of it) must be included. A possible refinement of the above rule is the algorithmic rule for typing conditionals in the presense of subtyping with calculated joins:

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \qquad \mathtt{T}_1 <: \mathtt{Bool} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2 \qquad \Gamma \vdash \mathtt{t}_3 : \mathtt{T}_3 \qquad \mathtt{T} = \mathtt{T}_2 \vee \mathtt{T}_3}{\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}} \qquad \text{T-IF } [\textit{bool}, \textit{calcjoin}]$$

We partition the repository's set of clauses into subsets sharing the same label. For each partition, we define a partial order dominance relation between its clauses based on the dominance relation between their corresponding sets of features.

For a repository to be consistent, we require that the dominance relation between clauses include the refinement relation. To some degree, this ensures that the content of clauses reflects the features that the clauses claim to implement.

Feature constraints are arbitrary propositional formulas over features. They enable us to verify that a system specified by a set of features makes sense and can actually exist. A system identified by features F is *consistent* if $\mathit{closure}(F)$ satisfies every formula in FC. For example, consider the reduction and evaluation relations on terms. Suppose [why??] that every type system must have either one or the other but not both. We can introduce features *normalize* and *eval* and add the formula $\mathit{normalize} \oplus \mathit{eval}$ to FC ($\oplus$ denotes "exclusive or") to achieve the desired condition.

Syntactic categories are sets of basic entities of a formal system. For example, in the realm of type systems some of the syntactic categories are terms, types, and contexts. Judgments represent relations between the elements of the syntactic categories. Depending on its features, a system may or may not contain certain judgements. This fact is formalized by the judgement enabling function that associates each judgement with a set of features. A system identified by features F contains a judgement $j \in J$ if F dominates $JC(j)$.

A given judgement may have different "shapes" in different formal systems. (For example, the subtyping judgement in the simply typed lambda-calculus is a two-place relation on types, $\texttt{S <: T}$; in $F_{<:}$ it is a three-place relation between contexts and pairs of types, $\Gamma \vdash \texttt{S <: T}$. To track these variations in shape (and prevent mixing rules of different shapes), we introduce judgement signatures. A signature has to mention the name of the judgements and the syntactic categories of the underlying relation. We call this pair of a judgement label and a set of syntactic category labels a judgment shape and write it $j(s_1, \ldots, s_n)$. This information alone is not enough to identify a judgement kind. For instance, the same syntactic categories form typing relation in systems with and without kinding. The type system of the clause assembler has to separate these judgements to be able to prevent mixing of clauses defining these two typing relations in the same output system. The distinguishing characteristic in this example is that the typing relation of the kindless system depends only on itself while the typing relation in a system with kinds depends on the kinding relation as well. Thus, we define judgement signatures as a pair of a judgment shape and a set of judgement shapes, and we write signatures using the arrow notation as follows:

$$j_1(s_{11}, \ldots, s_{1k}) \times \cdots \times j_m(s_{m1}, \ldots, s_{mk}) \to j(s_1, \ldots, s_n)$$

We call the shape to the right of the arrow the head. It describes the judgement which depends on the judgements represented by the left hand side shapes. For example, the typing judgement signature of a type system with kinds mentioned above may be:

$$\mathit{Typing}(\Gamma, \texttt{t}, \texttt{T}) \times \mathit{Kinding}(\Gamma, \texttt{T}, \texttt{K}) \to \mathit{Typing}(\Gamma, \texttt{t}, \texttt{T}),$$

where Γ is the set of contexts and $\texttt{t}$ and $\texttt{T}$ represent the term and type syntactic categories respectively. Finally, the clause signature mapping has the type $\mathit{CS} \in \mathit{CLS} \to \mathit{JS}$, where JS is the set of judgement signatures. The mapping assigns a judgement signature to every clause in the repository.

Judgement signatures enable two useful consistency checks—a global check on the whole repository and a local check on a particular system being assembled.

The global consistency condition ensures that no stray clauses defining irrelevent judgements can appear in generated systems. The assembler extracts the judgement signature from every clause in the repository and checks that the set of features enabling the signature's head judgement is dominated by the set of features relevant to the clause:

$$\forall (c \in \mathit{CLS}).\ \mathit{Features}(c) \text{ dominates } \mathit{JC}(\mathit{Head}(\mathit{CS}(c))),$$

where *Features* extracts relevant features of a clause, and *Head* returns the name of the head judgement given a judgement signature [maybe we can use pattern matching notation or something to keep from introducing all these aux functions.] This consistency check is a safeguard against a clause being mislabeled with a wrong relevant feature set.

The local consistency condition prevents clauses for different versions of the same judgement from ending up in the same output system, thus helping ensure that the generated system is well formed. It is acomplished by requiring that clauses of the generated system s with the same head judgement in their signature have the same signature:

$$\forall (c_1, c_2 \in s).\ (\mathit{Head}(\mathit{CS}(c_1)) = \mathit{Head}(\mathit{CS}(c_2)) \text{ implies } \mathit{CS}(c_1) = \mathit{CS}(c_2)).$$

This check alerts the user when a judgement for a feature required for building the output system has not been implemented in the repository. For example, suppose we want to build a system with kinding and universal polymorphism represented by features kinding and all respectively, but we forgot to implement the clause with the appropriate typing rule for the type application construct:

$$\frac{\Gamma \vdash \texttt{t}_1 : \forall \texttt{X}{::}\texttt{K}.\texttt{T}_1 \qquad \Gamma \vdash \texttt{T}_2 :: \texttt{K}}{\Gamma \vdash \texttt{t}_1\texttt{[T}_2\texttt{]} : \{\texttt{X} \mapsto \texttt{T}_2\}\texttt{T}_1} \qquad \text{T-TAPP } [\mathit{all}, \mathit{kinding}]$$

In the absense of the system well-formedness consistency check, the assembler will select the kindless version of the clause

$$\frac{\Gamma \vdash \mathtt{t}_1 : \forall \mathtt{X}.\mathtt{T}_1}{\Gamma \vdash \mathtt{t}_1[\mathtt{T}_2] : \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{T}_1} \qquad \text{T-TApp } [\textit{all}, \textit{simple}]$$

since it is the closest fit to the given feature specification assuming that *kinding* implies *simple*. Adding this rule to the result will produce an unsound type system since checking well-kindedness of the type parameter is essential.

We conclude this section by defining a term that describes the shape of a system: the *system signature* is the set of distinct judgement signatures of the system's clauses. We will use this term in a later chapter to classify features used to build type systems of various lambda calculi.

4 The TinkerType System

This section describes the TinkerType tool based on the ideas presented above. We have successfully applied the tool to build chapters of a book on type systems. One of the goals was to run examples in every chapter through the type checker and interpreter corresponding to this chapter and insert the output directly in the book. The book considers over 30 different type systems, and our tool helped tremendously in keeping them in a consistent state.

We represent clause contents by arbitrary strings. For instance, clauses containing bits of ML code can form a running type checker or interpreter; clauses with bits of TeX source can define a declarative presentation of a system. Consider the following part of the typechecking function `typeof`. This clause corresponds to T-If [*bool*, *simple*], the simple typing rule for conditional expressions mentioned before.

```
T-If
  {#TmIf(fi,s1,s2,s3) →
      if tyeqv ctx (typeof ctx s1) TyBool then
        let tyS = typeof ctx s2 in
        if tyeqv ctx tyS (typeof ctx s3) then tyS
        else error fi "arms of conditional have different types"
      else error fi "guard of conditional not a boolean"#}
```

Here, `T-If` is the label of the clause, and the content appears between the brackets `{#` and `#}`. To specify a refinement relation between two versions of a clause, we delimit the new bits of text in the refined clause by another set of brackets. (The concrete syntax choice for these brackets interacts with the object language syntax and is configurable.) Consider implementation of T-If [*bool*, *calcjoin*], the version of the clause above in the presense of subtyping:

```
T-If
  {#TmIf(fi,s1,s2,s3) →
      if [[[subtype]]] ctx (typeof ctx s1) TyBool then
        [[[join ctx (typeof ctx s2) (typeof ctx s3)]]]
      else error fi "guard of conditional not a boolean"#}
```

The text enclosed in the brackets `[[[` and `]]]` is new; while, the three pieces outside of the brackets occur consequtively in the body of the first clause. Whenever the latter condition holds of a pair of clauses, we say that the content of the second clause refines that of the first one.

As you may have noticed, the above two clauses do not declare their relevant features. In fact, having a list of features next to each clause would be too cluttering. Instead, we introduce the notion of *component*, a group of clauses relevant to the same set of features. For better structuring, we group clauses into nested sections. For example, we may have a `core` section for type checking related functions. Inside it we may have sections for individual functions, and they, in turn, may contain the actual clauses. The following fragment is a part of the [*bool*, *simple*] component:

```
core {
  tyeqv {
    eqv-bool {# ... #}
  }
```

```
  typeof {
    t-false {# ... #}
    t-true {# ... #}
    t-if {# ... #}
  }
}
```

Each section may define special clauses called `header`, `footer`, and `separator`. When the tool prints a section, it outputs the header, the section's subitems separated by the separator and followed by the footer. The subitems are printed in the order they appear in the components. Additionally, the user can explicitely specify an ordering constraint for one item relatively to another when they appears in different components.

The tool is run w.r.t. a *template* which could be a directory or a file. The template's files may contain hooks that refer to particular sections or clauses. The eventual goal is to make a copy of the template with each hook replaced by the output produced by the referenced item.

The tool operates as follows. Given a list of features, the tool figures out the relevant components, makes the necessary consistency checks, and merges the components by combining items that belong to the same sections. Then it traverses the template w.r.t. the combined component, instantiates the hooks, and outputs the final result.

For more flexibility we introduce macros as special code that can appear as part of a clause content. There is a macro that allows optional inclusion or exclusion of text depending on a command line flag or internal attribute. Another macro provides support for inclusion of text generated by other clauses or sections. We refer to this feature as parametrization. A use of this kind of macro is similar to a virtual method call in Java since the inclusion operation has late binding semantics; the refered item is selected from the combined component after merging is complete.

Another useful feature our tool has is optional omission of clauses repeated from a preceeding system. For example, we may output a system of simple types immediately followed by its extension with booleans. The latter system can be partially printed showing only the clauses specific to the boolean feature and omitting the ones that have not changed from the original system. In general, when printing these new bits of text, either brand-new clauses or new parts in refined clauses, the tool can delimit them by a user-defined prefix and suffix. We use this feature to highlight the new parts in the TeX output; you can see this in action in the previous section's example of T-If $[bool, calcjoin]$.

At the present stage, our tool does not fully implement judgement signature checking discussed in the previous section. We plan to incorporate it soon, but in the mean time, we provide a simpler consistency check that detects some of the problem covered by signature checking.

5 The Next 700 Type Systems

This section describes our current implementation of a number of type systems using the framework introduced above. We have built a repository of components, from which we generate about 40 different type systems. We support declarative and algorithmic presentations. The former consists of TeX code that implements rules of type systems; the algorithmic presentation consists of full fledged interpreters and type checkers written in ML. Below, we present our organisation of features and show how to use them to encode several interesting type systems including systems of Barendregt's lambda cube and a collection of lambda-calculi with subtyping. This section concentrates on the algorithmic systems, since their "feature skeleton" is more interesting.

5.1 Arrows, Booleans, and Subtyping Revisited

The remainder of this section discusses features and relationships between them. But firstly, to understand how components are built around features, we return to the example of boolean, subtyping, and arrow types presented in Section 2 and describe the list and content of the components needed to implement this example in our tool. Please, refer to Section 2 for the hierarchy of features used in the example.

We break components informally into two groups: infrastructure and content. The former components provide the basis for various judgements: they define auxiliary functions, the headers and signatures of

the functions that implement the judgements; provide headers of datatype declarations etc. The content components define clauses of the judgements for particular type constructors. Now, we list the components used in the example. Each component is indexed by its relevant features.

Infrastructure components:

- [*simple*]: introduction of the term and type datatypes and the typing judgement
- [*sub*]: introduction of the subtyping judgement
- [*calcjoin*]: introduction of the meet and join judgements

Content components:

- [*tmvar*]: declaration of the variable clause of the term datatype and simple typing rule for variables
- [*arrow*]: declaration of the abstraction and application clauses of the term datatype, arrow clause of the type datatype, and simple typing rules for abstraction and application
- [*arrow*, *sub*]: refinement of the typing rule for application with subtyping; subtyping rule for arrow types
- [*arrow*, *calcjoin*]: rules for meets and joins of arrow types
- [*bool*]: declaration of the false, true, and if clauses of the term datatype, bool clause of the type datatype, and simple typing rules for boolean constants and conditionals
- [*bool*, *sub*]: trivial subtyping rule for Bool
- [*bool*, *calcjoin*]: refinement of the typing rule for conditionals with subtyping and joins, plus trivial rules for meets and joins involving type Bool

5.2 Features for Variables and Binders

Identification of features and specification of dependencies and constraints between them is a complex process driven by the intricacies of the type systems we set out to model. To a large degree complexity of type systems arises from operations on term and type variables. We use DeBruijn notation to represent variables in ML code and provide customized shifting and substitution operations on types and terms. Since, for a particular type system, our goal is to make code as simple as possible and avoid unnecessary functionality, we must take into account whether type and term variables are present in the system, whether type and term variables appear in terms and types respectively, and which shifting and substitution operations are required in the system.

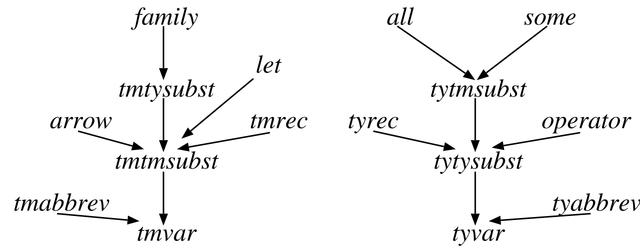


Figure 3: Variable and binding features

We start overview of our design by presenting a hierarchy of features related to operations on variables and binders. Figure 3 shows these features and relationships between them. Features with no incoming dependency arrows represent various variable binding forms. To some extent, the type and term variable

features are parallel to each other. Features *arrow* and *family* stand for abstraction of terms over terms and types over terms respectively. They correspond to *operator* and *all* in the type variable part of the hierarchy which stand for abstraction of types over types and terms over types respectively. Feature *let* provides local definitions and loosely corresponds to existential types represented by *some*. The *tmrec* and *tyrec* features give us recursive terms and types. The *variant* feature adds variant types and a branching construct over them. Finally, *tmabbrev* and *tyabbrev* define top level abbreviations.

Usually, features represent some kind of optional functionality. Thus, we do not need features for term and type shifting since these operations are always required when the corresponding variable features are present. Various kinds of substitution, on the other hand, may or may not be needed depending on the kind of type system one wants to build. Therefore, we created features *tmtmsubst*, *tmtysubst*, *tytmsubst*, and *tytysubst* that represent substitution of terms inside terms, terms inside types, types inside terms and types inside types respectively.

There are some features that correspond to real aspects we use to specify type systems; while some other features carry only technical significance to facilitate sharing of component code between different systems. The substitution features presented here are of the latter kind. It does not make sense to specify a system by *tmtysubst* since substitution is only useful inasmuch as some higher level feature, like *all*, requires it. So, generally, we create systems only from high level features.

In a straightforward implementation, the functions for different kinds of substitution can be encoded as inductive recursive functions over the shape of the parameter into which the substitution is made. That will require providing clauses for every kind of substitution for every relevant content feature. These clauses will be defined in new components tagged by relevant features $[tytmsubst, X]$ and $[tmtmsubst, X]$ where X ranges over $\{arrow, nat, bool, tmvar, \text{etc.}\}$ and $[tmtysubst, X]$ and $[tytysubst, X]$, where X ranges over $\{tyvar, all, operator, \text{etc.}\}$. To avoid prolifiration of components and clauses, we introduce general mapping functions over terms and types enabled by features *tmvar* and *tyvar* respectively. Then, we introduce components with relevant features $[tmvar, X]$ where X ranges over $\{arrow, nat, bool, tmvar, \text{etc.}\}$ and components with relevant features $[tyvar, X]$, where X ranges over $\{tyvar, all, operator, \text{etc.}\}$, whose goal is to define corresponding clauses for the term and type mapping functions. We implement substitution and shifting functions via the mapping procedures thus avoiding introducing clauses for every kind of substitution/shifting function - content feature pair.

The variable related features contain the cornerstones for building systems of Barendregt's lambda cube: *arrow* is the origin of the cube, while *all*, *typeontype*, and *typeonterm* represent the three dimensions of the cube. We will postpone the discussion of the cube, however, until we examine some other ways of slicing the world of type systems.

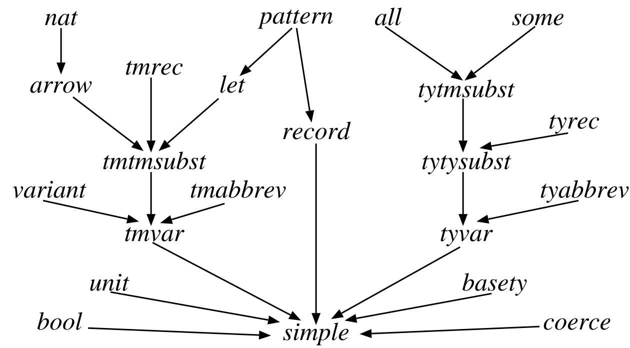
5.3 Simple Features

Another way to classify features is based on what kinds of judgements they support. The feature *simple* enables term and type formation and typing judgements. A large number of systems can be built containing these judgements and a choice of a reduction or evaluation relation on terms. We call these systems simple, and Figure 4 shows the hierarchy of features we use to build them.

The simple feature hierarchy includes almost all of the variable operation hierarchy and defines a variety of new content features. The feature *nat* represents natural numbers and the operation of iteration on them. The expression `iter n f b` returns `f (f ... (f b)...)`, where `f` is applied repeatedly `n` times. Because of the use of functions with `iter`, *nat* implies *arrow*. The feature *pattern* combines capabilities of local definition of *let* and record projection of *record* to provide pattern matching of record values. Features *bool* and *unit* represent corresponding types and operations on them; *basety* introduces base types viewed as just string constants; *coerce* defines explicit type casts.

5.4 Evaluation and Normalization Features

Any system built from the simple features can contain either a reduction and normalization or evaluation relation (with a particular reduction strategy selected) on terms but not both. To enforce this restriction, we introduce features *normalize* and *eval* and define a feature constraint $normalize \oplus eval$. This constraint requires one of these features to be explicitely added to a specification of every simple system. For systems



some

Figure 4: Simple features

with effects, a particular reduction strategy must be selected. Therefore, they can only support the evaluation relation. The feature *eff* provides the needed infrastrcture to implement effects, and *ref* is built on top of it to support reference cells.

5.5 Kinding Features

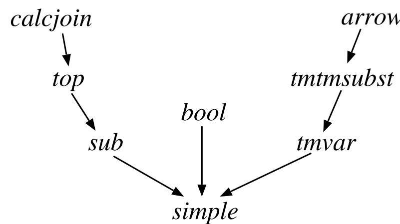


Figure 5: Kinding features

Systems with non-trivial type well-formedness relation introduce a kinding judgement. Type operators and dependent type families are both examples of systems with kinding. Our feature hierarchy is shown in Figure 5. Feature *kinding* enable the infrastructure of kinding. Feature *dep* introduces dependent functions and mutual dependency between the typing and kinding relations. Features*operator* and *family* define type operators and type families respectively.

5.6 Type Conversion Features

Syntactically equal types are considered equivalent in every type system. Additionally, three circumstances can cause syntactically different types to be viewed as equivalent. The types can be beta-convertible to

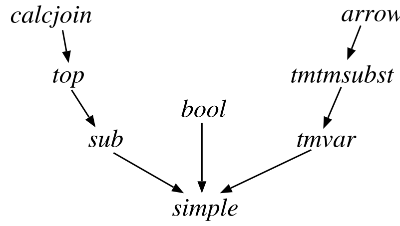


Figure 6: Type conversion features

each other in systems with *operator* or *family* features, or a type variable can be dereferenced to another type in the presense of the type abbreviation feature *tyabbrev*. When there is no subtyping, these features control whether to use syntactic equality or convertability for type comparison. Feature *tyconvert* (shown in Figure 6) signals presense of any one of the above three features and triggers the use of convertability; otherwise, the simple type equality is sufficient.

Now we have defined all the necessary features for building systems of Barendregt's lambda cube:

- $[arrow]$: simple types
- $[arrow, all]$: System F
- $[arrow, family]$: types dependent on terms
- $[arrow, all, operator]$: System F^{ω}
- $[arrow, all, operator, family]$: calculus of constructions

Features *all*, *operator*, and *family* define three different dimensions of the cube, while, *arrow* is the origin point.

5.7 Subtyping Features

A major difficulty in the design of systems with subtyping arises from the treatment of joins of types required for typing multi-armed conditional or variant branch expressions. There is a straightforward algorithm that calculates the join (least common supertype) of an arbitrary pair of types in systems that have both joins and meets. Not all systems are like that however. For example, full fsub is a system that contains pairs of types that have no join. Applying the above algorithm in such a system is unsound. The solution is to provide the algorithm for calculating joins in systems with joins and meets but require annotation of multi-armed expressions with the intended result type in systems without joins. We achieve this by introducing mutually exclusive *calcjoin* and *anotjoin* features. Whenever a system contains any of the multi-armed forms, it has to have either *calcjoin* or *anotjoin*; hence, the repository contains the following feature constraint:

$$bool \lor variant \Rightarrow calcjoin \oplus anotjoin$$

Mutually exclusive *kfsub* and *ffsub* stand for kernel and full versions of $F_{<:}$. The corresponding constraint is

$$\neg(kfsub \land ffsub).$$

Finally, *ffsub* excludes the possibility of having calculated joins:

$$ffsub \Rightarrow \neg(calcjoin)$$

Figure 7 presents the subtyping hierarchy where *boundedtyvar* represents bounded type variable declaration and is implied by any of the fsub features. Among others, we can build the following subtyping systems from the presented features:

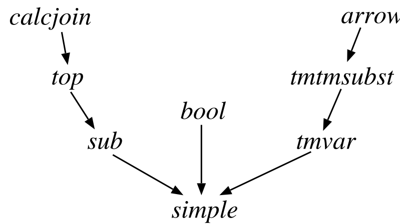


Figure 7: Subtyping features

- [*arrow*, *bool*, *calcjoin*]: simple subtyping with booleans and calculated joins
- [*arrow*, *all*, *kfsub*]: kernel $F_{<:}$
- [*arrow*, *all*, *ffsub*]: full $F_{<:}$
- [*arrow*, *all*, *operator*, *kfsub*]: kernel $F^{\omega}_{<:}$
- [*arrow*, *all*, *operator*, *ffsub*]: full $F^{\omega}_{<:}$

Our components are tuned to recognize whether *boundedtyvar* is present in a system and define the subtyping judgement with or without variable context depending on that. We achive it by refining every [$\star$, *sub*] component with a corresponding [$\star$, *boundedtyvar*] component that adds a context to every applicable clause.

5.8 Parameterization

To avoid copying the same code in many places, our components judiciously use parametrization described in the previous section. A characteristic example of this is the way we implement type comparison in type checking algorithms. Depending on the system, three different type comparison operations can be used: syntactic equality, convertability, or subtyping. Many clauses of the type checking algorithm compare types: application, if expression, record projection clauses etc. We can provide three different versions of each one of them for the case of equality, convertability, and subtyping, but that would be too verbose. Instead, we introduce an auxiliary clause called TYCMP-PROC, whose content is the name of the comparison procedure, and include it (using the macro facility described in the previous section) wherever types are compared. That way we have to provide three different versions of TYCMP-PROC, but only one version of any clause that compares types (unless it has to be refined for other reasons.)

6 Related Work

The initial inspiration for our work came from the *type system fragments* used by Abadi and Cardelli in their book, *A Theory of Objects* [AC96]. There, the repository consists of a collection of named "fragments" analogous to our components. A system is specified by naming a collection of fragments whose contents are to be concatenated. This is a degenerate instance of our framework, where each component is labeled with a single, distinct feature and where there is no dependency between features, and where no static consistency checks are performed. Their book presents a substantial collection of fragments, covering (declarative formulations of) approximately the same range of type systems as the ones described here in Section 5.

Another close relative of our work is Prehofer's *feature-oriented programming* [Pre97]. Like our approach, it includes features and dependencies between them, components with multiple variants, and an assembly process that combines appropriate variants based on a set of requested features. The main difference is the application domain. Our approach focuses on formal systems, and the basic unit of composition is an individual inference rule. Feature-oriented programming is used to assemble objects; its basic unit of

composition is a group of related methods. Prehofer introduces an extension of Java with feature support and describes two approaches for compiling it into Java.

The *Hyperspace* project [OT99, TOHS99] introduce a general theory of multi-dimensional separation of concerns. In their work, *units* are atomic entities similar to our clauses. A unit can be related to several *concerns*, which correspond to our features. Concerns are partitioned into orthogonal *dimensions*. *Hyperslices* are composed of units and resemble our components. They can be merged to form *hypermodules* that are similar to systems in our work. This approach is somewhat more abstract than ours (for example, the algorithm for merging hyperslices is taken as a parameter).

Earlier work in the same group promoted a technology called *subject-oriented programming* [HO93]. One of its principal goals was to allow parallel development of classes and provide a composition mechanism to obtain a final system. In this view, classes resemble our components, and their merging is analogous to system assembly. No mechanism corresponding to features is provided.

Aspect-oriented programming [KLM+97, Kic96] starts from the observation that it is sometimes difficult to address certain issues in a programs without obscuring its main functionality. These issues, called *aspects*, "cross-cut" the natural decomposition of the main functionality, resulting in small bits of related code strewn across the system. To simplify designing programs with these properties, AOP proposes using conventional *component languages* to implement basic functionality, and special purpose *aspect languages* to deal with the cross-cutting issues. A special process called *weaving* merges programs written in these languages to produce the resulting system. To some extent, we can view our language of features, clauses, and components as a particular aspect language; the component language is whatever language is used to express the contents of clauses.

Another area of related work is monadic techniques for structuring interpreters and compilers [Ste94, LHJ95, Esp95, LH96, HK98, etc.]. The focus here is on modular definition and combination of different aspects of computation (state, exceptions, concurrency, etc.). It is a highly structured approach, using the type system of the metalanguage to control the composition process and focusing on constraints arising from interaction between features. It does not appear easy to extend the monadic approach to typing features in the spirit of the present work. On the other hand, we believe that a monadic style could be used to structure the presentation of the operational semantics of our typed lambda-calculi.

7 Conclusions and Future Work

Current status

- how we generate the book (examples get checked every time, etc.)
- 9,000 lines of TinkerType sources,
- generating 67,000 lines of typecheckers (and plenty of TeX)

Pros of our approach (compared to e.g. PTS)

- leafiness
- control over pedagogical aspects
- ability to design new systems of classification at will

The framework can potentially be applied in a wide variety of settings: anyplace where we're interested in programs that can be broken down into clauses (i.e., that consist of definitions by cases / pattern matching).

Cons: harder to reason "uniformly" about the formal systems we generate (the usual problem with reasoning in the presence of inheritance: all reasoning must be "parametric in all possible extensions")

Future work:

- Modular metatheory proofs (mention preliminary experiments),
- More static consistency checking

Our approach to formal systems is rather unstructured: we treat the contents of clauses as uninterpreted strings and perform only rudimentary checks on their signatures. (For example, we completely ignore issues of variable binding.) The payoff from this lack of structure is considerable flexibility, but we would like to see how much structure could be added without impeding malleability. Recent work by Fiore, Plotkin, and Turi on a general treatment of abstract syntax and variable binding [FPT] may help point the way.

Acknowledgments

This work was supported by the University of Pennsylvania and by NSF grant CCR-9701826, *Principled Foundations for Programming with Objects*.

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[Bar92] Henk Barendregt. Lambda calculi with types. In Gabbay Abramsky and Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

[BMV95] David Basin, Seán Matthews, and Luca Viganò. A modular presentation of modal logics in a logical framework. In *Proceedings of the Tbilisi Symposium on Language, Logic and Computation*, October 1995.

[Esp95] David Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995.

[FPT] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. Preprint, available through `http://www.dcs.ed.ac.uk/~dt/abstractsyn.ps`.

[Gun92] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. The MIT Press, Cambridge, MA, 1992.

[HK98] William L. Harrison and Samuel N. Kamin. Modular compilers based on monad transformers. In *Proceedings of the IEEE International Conference on Computer Languages*, 1998.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In Andreas Paepcke, editor, *OOPSLA 1993 Conference Proceedings*, number 28(10) in ACM SIGPLAN Notices, pages 411–428. ACM Press, October 1993.

[Jac99] Bart Jacobs. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Elsevier, 1999.

[Kic96] Gregor Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es):154, December 1996.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in Lecture Notes in Computer Science. Springer-Verlag, June 1997.

[Lan65] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Parts I and II. *Communications of the ACM*, 8(2,3):89–101, 158–165, February and March 1965.

[Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[LH96] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *Programming Languages and Systems – ESOP'96, Proc. 6th European Symposium on Programming, Linköping*, volume 1058 of *Lecture Notes in Computer Science*, pages 219–234. Springer-Verlag, 1996.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995*, pages 333–343, New York, NY, USA, 1995. ACM Press.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, Cambridge, MA, 1996.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Fourth Annual Symposium on Logic in Computer Science (Asilomar, CA)*, pages 14–23. IEEE Computer Society Press, June 1989.

[Mos92] Peter D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[OT99] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns in hyperspace. Technical Report RC 21452(96717)16APR99, IBM T.J. Watson Research Center, 1999.

[Pie] Benjamin C. Pierce. Type systems. Draft book; publication projected for 2001.

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443, Jyväskylä, Finland, 9–13 June 1997. Springer.

[SOW97] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL 4)*, January 1997. Full version in *Theory and Practice of Object Systems, 1998*.

[Ste94] Guy L. Steele, Jr. Building interpreters by composing monads. In ACM, editor, *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: Portland, Oregon, January 17–21, 1994*, pages 472–492, New York, NY, USA, 1994. ACM Press.

[TOHS99] Peri Tarr, Harold L. Ossher, William H. Harrison, and Stanley M. Sutton, Jr. N degrees of separation: Multi-dimentional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.